COMP0010 Coursework Project

Sergey Mechtaev
University College London
s.mechtaev@ucl.ac.uk

Module code	COMP0010	**Module title**	Software Engineering	
Assignment title	JSH: Reengineering a legacy shell			
Weighting in module	100% (50% for group work, 50% for individual work)		**Report length**	Maximum 3 pages
Set by	Dr Sergey Mechtaev		**Moderated by**	Sarah Sanders
Deadline	06/01/2020 12:00pm (midday)			
Submission method	Electronic submission to Moodle		**Feedback method**	Turnitin/Moodle
Marks & feedback Provisional marks and feedback will normally be returned within one calendar month. Students will be advised via Moodle of any delay. Unless otherwise stated above feedback will be provided via the Moodle TurnItIn.				

TABLE OF CONTENTS

INTRODUCTION

You have joined a startup that develops software for IoT devices. This software runs on a custom low power consumption operating system. The main user interface of this system is a homegrown shell called JSH. Since its inception, the company relies on this shell, and

many scripts and commands have been already written by the users and developers of the system. However, the functionality and stability of the shell do not longer meet the requirements of the growing business. At the same time, legacy code makes it too expensive to switch to a different shell implementation. Your manager asks you and two other novice developers to extend the capabilities of JSH, while preserving the original features and fixing existing bugs. After examining the source code of JSH, you and your colleagues realize that it was developed without following software engineering practices, and as a result, it is extremely difficult to extend the implementation without introducing new bugs which will lead to problems in production. Thus, the first step is to refactor the legacy implementation to make it extendable, and add a regression test-suite to avoid functionality-breaking changes. You receive the source code of the shell, and an incomplete specification of its functionality, as well as a description of the desired extension. In order to complete this task, you will need to apply the principles and techniques that you learned from COMP0010 Software Engineering course.

BACKGROUND

A shell is a command interpreter. Its responsibility is to interpret commands that the user type and to run programs that the user specifies in his/her command lines. The Figure 1 shows the relationship between the shell, the kernel, and applications in a UNIX-like operating system.

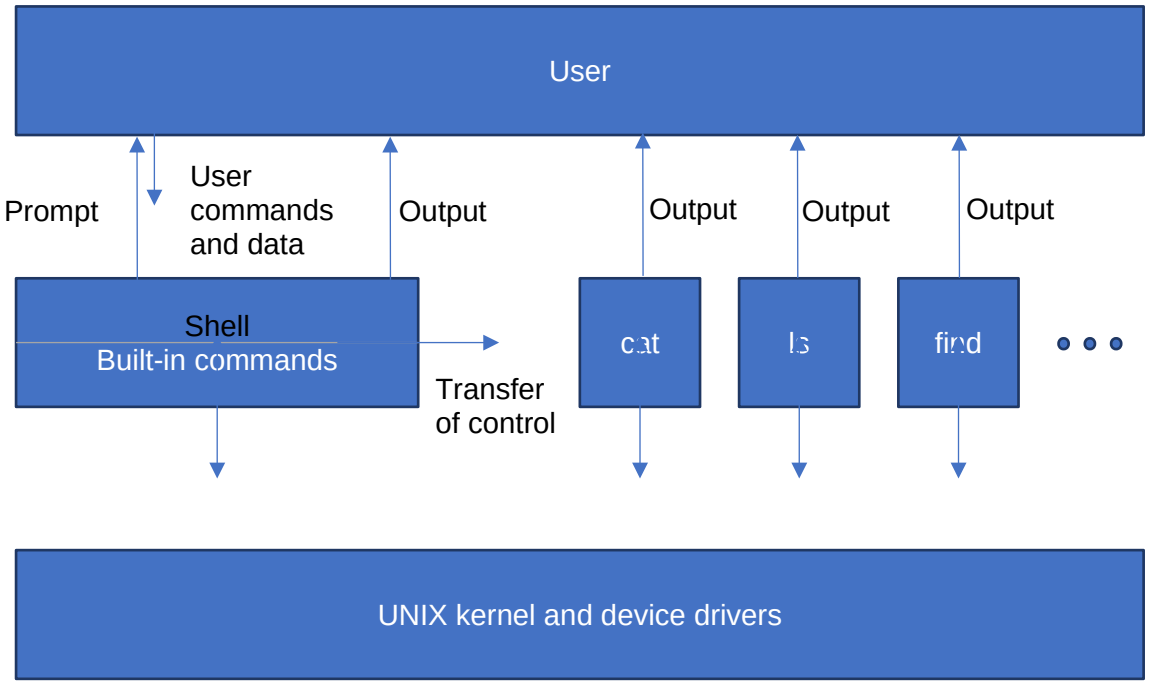

Figure 1

Shell can be thought of as a programming language for running applications. From the user's perspective, it performs the following loop:

1. Print prompt message.
2. Parse user's command.

3. Interpret user's command, run specified applications or built-in commands.
4. Print output.
5. Go to 1.

A command line application can be considered as a block with two inputs and three outputs (Figure 2). Such application reads input data from its standard input stream (stdin) and a of command line arguments. Then, it writes output data to its standard output stream (stdout) and error information to its standard error stream (stderr). After execution, the application also returns exit status, a number that indicates an execution error if it is non-zero.

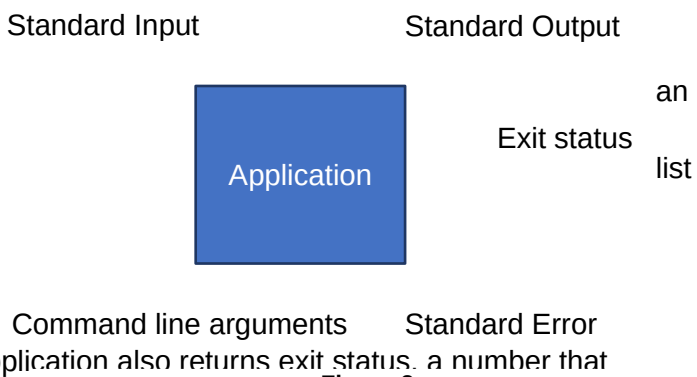

Figure 2

An important functionality of shells in UNIX-like systems is the ability to compose complex commands from simple ones. For example, the following command combines “cat” and “grep”:

```
cat articles/* | grep “Interesting String”
```

Shell expands “articles/*” into the list of files in the directory “articles“, and passes them to “cat” as command line arguments. Then, “cat” concatenates the contents of all these files and passes the result to “grep” using the pipe operator “|” (Figure 3). “grep” finds and displays all the lines that include “Interesting String” as a substring.

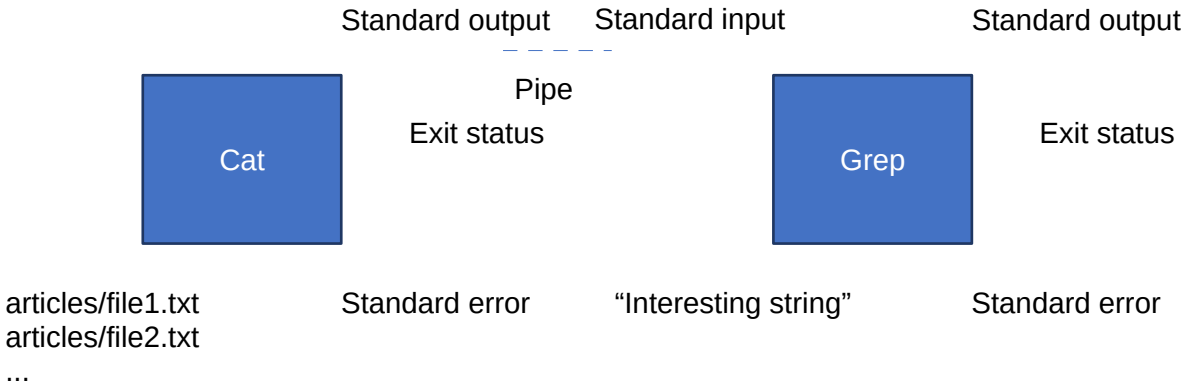

Figure 3

Grammar notation

We use BNF (Backus-Naur form) to describe the shell language, that is how the shell parses user’s commands. The notation consists of a set of rules. For instance, the following rule states that a command is a pipe command, or a sequence command, or a call command:

```
<command> ::= <pipe> | <seq> | <call>
```

The following rule states that a sequence command is a string consisting of two commands separated by semicolon:

```
<seq> ::= <command> “;” <command>
```

The asterisk operator indicates that the operand can be repeated any number of times (including zero), while the plus operator means that the operand has to be repeated at least one time. For instance, the following rule states that a call command is a string consisting of zero of more occurrences of either non-keyword symbol or a quoted string:

```
<call> ::= ( <non-keyword> | <quoted> ) *
```

The following rule states that an input redirection is a string that starts with the less symbol, followed by an optional whitespace, and then by an argument:

```
<redirection> ::= “<” [ <whitespace> ] <argument>
```

SPECIFICATION

The goal of this project is to extend test a shell and a set of applications. shell, JSH, and the applications are implemented in Java. The required functionality is a subset (or simplification) of the functionality provided by shell used in UNIX-like systems. Particularly, the specification resembles the behavior of Bash shell. However, there are several important distinctions:

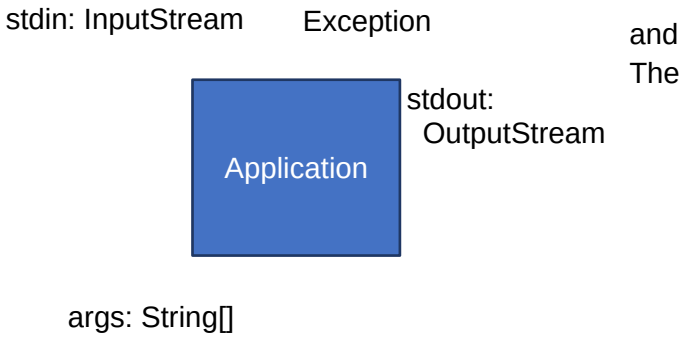

Figure 4

1. JVM is used instead of OS kernel/drivers to provide required services.
2. Shell and all applications are run inside the same process.
3. Applications raise exceptions instead of writing to stderr and returning non-zero exit code in case of errors.

A shell can be thought of as a programming language where applications play the same role as functions in C and JAVA. Shell parses user's command line to determine the applications to run and the data to pass to these applications. JSH supports two ways to specify input data for applications: by supplying command line arguments and by redirecting input streams.

Command line parsing

User's command line can contain several subcommands. When JSH receives a command line, it performs the following:

1. Parses the command line on the command level. JSH recognizes three kind of commands: call command, sequence command, and pipe command.
2. The recognized commands are evaluated in the proper order.

Step 1 uses the following grammar:

```
<command> ::= <pipe> | <seq> | <call>
<pipe> ::= <call> “|” <call> | <pipe> “|” <call>
```

```
<seq>  ::= <command> “;” <command>
<call> ::= ( <non-keyword> | <quoted> ) *
```

A non-keyword character is any character except for newlines, single quotes, double quotes, backquotes, semicolons “;” and vertical bars “|”. The non-terminal <quoted> is described below.

Quoting

To pass several arguments to an application, we can separate them by spaces:

```
echo hello world
```

In this example, “echo” gets two command line arguments: “hello” and “world”. In order to pass “hello world” as a single argument, we can surround it by quotes, so that the interpretation of the space character as a separator symbol is disabled:

```
echo “hello world”
```

In this case, “echo” receives “hello world” as a single argument (without quotes).

JSH supports three kinds of quotes: single quotes ('), double quotes (“) and backquotes (`). The first and the second ones are used to disable interpretation of special characters, the last one is used to make command substitution.

JSH uses the following grammar to parse quoted strings:

```
<quoted>        ::= <single-quoted> | <double-quoted> | <backquoted>
<single-quoted> ::= “'” <non-newline and non-single-quote> “'”
<backquoted>    ::= “`” <non-newline and non-backquote> “`”
<double-quoted> ::= “”” ( <backquoted> | <double-quote-content> ) *
“”””
```

where <double-quote-content> can contain any character except for newlines, double quotes and backquotes.

Note that the rule for double quotes is different: double quotes do not disable interpretation of backquotes. For example, in the following command:

```
echo “this is space: `echo “ ”`”
```

the outer “echo” receives one argument rather than two.

Note that, compared with e.g. Bash, we do not use character escaping in JSH.

Call command

Example

```
grep “Interesting String” < text1.txt > result.txt
```

Find all the lines of the file text1.txt that contain the string “Interesting String” as a substring and save them to the file result.txt.

Syntax

JSH splits call command into arguments and redirection operators.

```
<call> ::=
  [ <whitespace> ] [ <redirection> <whitespace> ]* <argument>
  [ <whitespace> <atom> ]* [ <whitespace> ]
<atom> ::= <redirection> | <argument>
<argument> ::= ( <quoted> | <unquoted> )+
<redirection> ::= “<” [ <whitespace> ] <argument> |
                  “>” [ <whitespace> ] <argument>
```

Whitespace is one or several tabs or spaces. An unquoted part of an argument can include any characters except for whitespace characters, quotes, newlines, semicolons “;”, vertical bar “|”, less than “<” and greater than “>”.

Semantics

A call command is evaluated in the following order:

1. Command substitution is performed.
2. The command is split into arguments. The command string is split into substring corresponding to the <argument> non-terminal. Note that one backquoted argument can produce several arguments after command substitution. All the quotes symbols that form <quoted> non-terminal are removed.
3. Filenames are expanded (see globbing).
4. Application name is resolved (the first <argument> without a redirection operator).
5. Specified application is executed.

When JSH executes an application, it performs the following steps:

1. IO-redirection. Open InputStream from the file for input redirection (the one following “<” symbol). Open the OutputStream to the file for output redirection (the one following “>” symbol). If several files are specified for input redirection or output redirection, throw an exception. If no files are given, use the NULL value. If the file specified for input redirection does not exist, throw an exception. If the file specified for output redirection does not exist, create it.
2. Running. Run the specified application, supplying given command line arguments and redirection streams.

Semicolon operator

Example

```
cd articles; cat text1.txt
```

Change the current directory to articles. Display the content of the file text1.txt.

Syntax

```
<seq> ::= <command> “;” <command>
```

Semantics

Run the first command; when the first command terminates, run the second command. If an exception is thrown during the execution of the first command, the execution if the whole command must be terminated.

Pipe operator

Example

```
cat articles/text1.txt | grep “Interesting String”
```

Find all the line of the file articles/text1.txt that contain “Interesting String” as a substring.

Syntax

```
<pipe> ::= <call> “|” <call> |
           <pipe> “|” <call>
```

Semantics

Pipe is a left-associative operator that can be used to bind a set of call commands into a chain. Each pipe operator binds the output of the left part to the input of the right part, then evaluates these parts concurrently. If an exception occurred in any of these parts, the execution of the other part must be terminated.

Globbing

Example

```
cat articles/*
```

Display the content of all the files in the articles directory.

Syntax

The symbol * (asterisk) in an unquoted part of an argument is interpreted as globbing.

Semantics

For each argument ARG that contains unquoted * (asterisk) perform the following:

1. Collect all paths to existing files and directories such that these paths can be obtained by replacing all the unquoted asterisk symbols in ARG by some (possibly empty) sequences of non-slash characters.
2. If there are no such paths, leave ARG without changes.
3. If there are such paths, replace ARG with a list of these path separated by spaces.

Note that globbing (filenames expansion) is performed after argument splitting. However, globbing produces several command line arguments if several paths are found.

Command substitution

Example

```
wc -l `find -name '*.java'`
```

Find all files whose names end with “.java”, and count the number of lines in these files.

Syntax

A part of a call command surrounded by backquotes (`) is interpreted as command substitution iff the backquotes are not inside single quotes (see the non-terminal <backquoted>).

Semantics

For each part SUBCMD of the call command CALL surrounded by backquotes:

1. SUBCMD is evaluated as a separate shell command yielding the output OUT.
2. SUBCMD together with the backquotes is substituted in CALL with OUT. After substitution, symbols in OUT are interpreted the following way:
    a. Whitespace characters are used during argument splitting. Since JSH does not support multi-line commands, newlines in OUT should be replaced with spaces.
    b. Other characters (including quotes) are not interpreted during the next parsing step as special characters.
3. The modified CALL is evaluated.

Note that command substitution is performed after command-level parsing but before argument splitting.

Applications

Applications that require stdin stream cannot read it interactively as in Bash, but only use input streams provided by the shell through redirections. If expected stdin is not provided, the application must throw an exception. If the required command line arguments are not provided or arguments are wrong or inconsistent, the application throws an exception as well.

The pwd application outputs the current working directory followed by a newline:

```
pwd
```

The cd application changes the current working directory:

```
cd PATH
```

- PATH – relative path to the target director.

The ls application lists the content of a directory. Prints a list of files and directories separated by tabs and followed by a newline. Ignores files and directories whose names start with ".":

```
ls [PATH]
```

- PATH – the directory. If not specified, list the current directory.

The cat command concatenates the content of given files and prints on the standard output:

```
cat [FILE]...
```

- FILE(s) – the name of the file(s). If no files are specified, use stdin.

The echo command writes its arguments separated by spaces and terminates by a newline on the standard output:

```
echo [ARG]...
```

The head application prints first N lines of the file (or input stream). If there are less than N lines, print only the existing lines without raising an exception:

```
head [OPTIONS] [FILE]
```

- OPTIONS – “head -n 15” means printing 15 lines.Print the first 10 lines if not specified.
- FILE – the name of the file. If not specified, use stdin.

Prints the last N lines of the file (or input stream). If there are less than N lines, print the existing lines without raising an exception.

```
tail [OPTIONS] [FILE]
```

- OPTIONS – “tail -n 15” means printing 15 lines. Print the last 10 lines if not specified.
- FILE – the name of the file. If not specified, use stdin.

The grep command searches for lines containing a match to the specified pattern. The output of the command is the list of the lines. Each line is printed followed by a newline:

```
grep PATTERN [FILE]...
```

- PATTERN – specifies a regular expression in JAVA format.
- FILE(s) – the name of the file(s). If not specified, use stdin.

The sed application copies input file (or input stream) to stdout performing string replacement. For each line containing a match to a specified pattern (in JAVA format), replaces the matched substring with the specified string.

```
sed REPLACEMENT [FILE]
```

- REPLACEMENT
  `s/regexp/replacement/`
  replace the first substring matched by regexp in each line with the string replacement.
  `s/regexp/replacement/g`
  replace all the substrings matched by regexp in each line with the string replacement. Note that the symbols “/” used to separate regexp and replacement string can be substituted by any other symbols. For example, “s/a/b/” and “s|a|b|” are the same replacement rules. However, this separation symbol should not be used inside the regexp and the replacement string.
- FILE – the name of the file. If not specified, use stdin.

The find command recursively searches for files with matching names. It outputs the list of relative paths, each followed by a newline:

```
find [PATH] -name PATTERN
```

- PATTERN – file name with some parts replaced with * (asterisk).
- PATH – the root directory for search. If not specified, use the current directory.

The wc command prints the number of bytes, words, and lines in given files (followed by a newline):

```
wc [OPTIONS] [FILE]...
```

- OPTIONS
  - -m : Print only the character counts
  - -w : Print only the word counts
  - -l : Print only the newline counts
- FILE(s) – the file(s), when no file is present, use stdin.

An unsafe version of an application is an application that has the same semantics as the original application, but instead of raising exceptions, it prints the error message on the standard output, and terminates successfully. The names of unsafe applications are prefixed with "_", e.g. "_ls", "_grep", etc.

REQUIREMENTS

The existing legacy implementation of JSH provides the following functionality:

- Shell: calling applications, quoting, semicolon operator, globbing.
- Applications: cd, pwd, ls, cat, echo, head, tail, grep.

Your goal it to preserve the existing functionality (while fixing possible bugs), and implement the following new features:

- Shell: pipe operator, IO-redirection, command substitution.
- Applications: find, wc, sed, unsafe versions of all applications.

You are also expected to refactor the current implementation to make it more modular and extendable. You are expected to write JUnit tests that would provide 97% branch coverage. The project must be analyzed with PMD, and all critical and blocker violations must be fixed.

PACKAGE

JSH deployment/submission package should contain the following files and directories:

- `src/`: source code and tests
- `pom.xml`: Maven build configuration
- `Dockerfile`: commands to build Docker image
- `jsh`: script to run JSH
- `test`: script to run unit tests
- `analysis`: script to run PMD (static analysis)
- `coverage`: script to run JaCoCo (code coverage)
- `.devcontainer/Dockerfile`, `.devcontainer/devcontainer.json`, `settings/*`, `.classpath`, `.project`: Visual Studio Code project files.

The package should provide the following interface:

Action	Command
Build JSH Docker image	`docker build -t jsh .`
Execute JSH in interactive mode	`docker run -it --rm jsh /jsh/jsh`
Evaluate shell command (e.g. echo foo) with JSH	`docker run -it --rm jsh /jsh/jsh -c 'echo foo'`
Run JUnit tests	`docker run -p 80:8000 -it --rm jsh /jsh/test`
Run PMD analysis	`docker run -p 80:8000 -it --rm jsh /jsh/analysis`
Run JaCoCo coverage	`docker run -p 80:8000 -it --rm jsh /jsh/coverage`

After running JUnit, PMD, or JaCoCo, the corresponding reports should be available at the following URLs:

Report	URL
JUnit	http://localhost/surefire-report.html
PMD	http://localhost/pmd.html
JaCoCo	http://localhost/jacoco

INDIVIDUAL REPORT

Your report should contain your individual reflection on the software engineering practices used in the project, and describe your individual contribution. Specifically, you could include:

1. Describe design principles and design patterns that you applied in the project. What problems did they solve? Did you consider alternative options and their trade-offs? What was the impact of these design choices on the overall result? What conclusions can you draw from your experience?
2. What approach to refactoring did you choose, and what lessons did you learn?
3. Did you manage to find interesting bugs using testing? What aspects of testing appeared to be more useful than the others? How does testing affect to your design?
4. Did you apply TDD and was it beneficial for the project?
5. Did you manage to find interesting bugs using static analysis?
6. How did you organize communication in your team? What lessons did you learn?
7. Describe other aspects related to software engineering.

Be concrete in your explanations (e.g. give references to your code), however put your reflections to wider context. Note that you should focus less on the shell and its specification, and more on development process, design, quality assurance, and other software engineering aspects that are covered in the course. Report both positive and negative experience.

By default, we assume that the group members contribute equally to the project. However, you might explicitly specify individual contributions in your report, e.g. “I contributed 40% and the other team members contributed 30% and 30%”, however this information needs to be consistent with the reports of the other team members.

SUBMISSION

Your submission should include two parts: (1) a submission package developed in groups of three, and (2) a report written individually. Before the deadline, you should do the following:

1. Prepare a submission package, ensure that the intended output of the **docker commands** specified in the section Package can be successfully replicated.
2. Ensure that your submission package can be **opened in Visual Studio Code** with Remote Development plugin (using the "open in a container" option).
3. Tag the version of the source code that you intend to submit with a **tag "final"** and push the tag to your GitHub Classroom repository, e.g.
   ```
   git tag final master
   git push origin final
   ```
4. One member of the group should submit the package corresponding to the tag "final" in your repository to Moodle. The submitted file should be a ZIP achieve named as **"<Index of your team>.zip",** e.g. for Team 23, the file should be named "23.zip".
5. Each student should write an individual report described in the section Individual report and submit via Turnitin on Moodle. The document must be in Arial pt. 11 (or equivalent font) with all margins set to 2-2.5cm and saved as a PDF. You must use the following naming convention for your file: **"<Student ID>.pdf"**, e.g. "18000000.pdf". The maximum length of the report is 3 pages. For submissions exceeding the specified maximum length, the mark might be reduced.